# Chapter 8
# Stochastic search methods for CSPs

## 8.1 Introduction

In many situations, a timely response by a CSP solver is crucial. For example, some CSPs may take days or years to solve with conventional hardware using the complete search methods so far introduced in this book. In applications such as industrial scheduling, the user may like to analyse a large number of hypothetical situations. This could be due to the fact that many factors are unknown to the user (who would like to explore many hypothetical situations), or that many constraints are merely preferences which the user is prepared to relax if no solution which satisfies them all can be found. For such applications, the user may need to evaluate the effect of different combinations of constraints, and therefore speed in a CSP solver is important.

In other applications, the world might be changing so dynamically that delay in decisions could be extremely costly. Sometimes, decisions could be useless if they come too late. For example, in scheduling transportation vehicles, in a container terminal, one may be allowed very little time to schedule a large number of vehicles and delays could be very costly. In allocating resources to emergency rescue teams, a decision which comes too late is practically useless.

Although linear speed up may be achievable with parallel architecture (architecture which use multiprocessors), it is not sufficient to contain the combinatorial explosion problem in CSPs. When no alternative methods are available, the user may be willing to sacrifice completeness for speed. (In fact, completeness is seldom guaranteed by human schedulers in the kind of applications mentioned above.) When this is the case, *stochastic search* methods could be useful.

Stochastic search is a class of search methods which includes heuristics and an element of nondeterminism in traversing the search space. Unlike the search algorithms introduced so far, a stochastic search algorithm moves from one point to

another in the search space in a nondeterministic manner, guided by heuristics. The next move is partly determined by the outcome of the previous move. Stochastic search algorithms are, in general, incomplete.

In this chapter, we introduce two stochastic search methods, one based on hill-climbing and the other based on a connectionist approach. Both of them are general techniques which have been used in problems other than the CSPs. We shall focus on their application to CSP solving.

## 8.2 Hill-climbing

*Hill-climbing* is a general search technique that has been used in many areas; for example, optimization problems such as the well known *Travelling Salesman Problem*. Recently, it has been found that hill-climbing using the min-conflict heuristic (Section 6.3.2 in Chapter 6) can be used to solve the *N*-queens problem more quickly than other search algorithms.[1] We shall first define the hill-climbing algorithm, and then explain its application to CSP.

### 8.2.1 General hill-climbing algorithms

The general hill-climbing algorithm requires two functions: an *evaluation function* which maps every point in the search space to a value (which is a number), and an *adjacency function* which maps every point in the search space to other points. The solution is the point in the search space that has the greatest value according to the evaluation function. (Minimization problems are just maximization problems with the values negated.)

Hill-climbing algorithms normally start with a random focal point in the search space. Given the current focal point *P*, all the points which are adjacent to *P* according to the adjacency function are evaluated using the evaluation function. If there exist some points which have greater values than *P*'s, then one of these points (call them "higher points") will be picked nondeterministically to become the new focal point. Heuristics can be used for choosing from among the higher points when more than one exists. A certain degree of randomness is often found to be useful in the selection. The algorithm continues until the value of the current focal point is greater than the values of all the nodes adjacent to it, i.e. the algorithm cannot climb to a higher point. The current focal point is then either a solution or a local maximum. The pseudo code for the generic hill-climbing algorithm is shown below:

---

1. Deterministic algorithms for solving the *N*-queens problem exist (e.g. see Abramson & Yung, 1989 and Bernhardsson, 1991)

```
PROCEDURE Generic_Hill_Climbing(e,c)
/* Given a point P in the search space, e(P) maps P to a numerical
   value which is to be maximized; c maps any point P to a (possibly
   empty) set of points in the search space*/
BEGIN
   /* Initialization */
   P ← random point in the search space;
   SP ← c(P);          /* SP is the set of points adjacent to P */

   /* Hill-climbing */
   WHILE (there exists a point Q in SP such that e(Q) ≥ e(P)) DO
      BEGIN
         Q ← a point in SP such that e(Q) ≥ e(P);
         /* heuristics may be used here in choosing Q */
         P ← Q;
         SP ← c(P);
      END
END /* of Generic_Hill_Climbing */
```

There are different ways to tackle a CSP with a hill-climbing approach. The following is the outline of one. The search space comprises the set of all possible compound labels. The evaluation function maps every compound label to the negation of the number of constraints being violated by it. (Therefore, a solution tuple will be mapped to 0.) Alternatively, the value of a compound label could be made the negation of the number of labels which are incompatible with some other labels in the compound label. The next step is to define the adjacency function *c* in the Generic_Hill_Climbing procedure. Two compound labels may be considered to be adjacent to each other if they differ in exactly one label between them. In the following we show the pseudo code of a naive hill-climbing algorithm for tackling CSPs. There the CSP is treated as a minimization problem in which one would like to minimize the number of constraints being violated. A solution to the CSP is a set of assignments which violates zero constraints:

```
PROCEDURE Naive_CSP_Hill_Climbing(Z, D, C)
BEGIN
   /* Initialization */
   CL ← { };
   FOR each x in Z DO
      BEGIN
         v ← a random value in D_x;
         CL ← CL + {<x,v>};
      END

   /* Hill-climbing: other termination conditions may be added to pre-
```

```
    vent infinite looping */
  WHILE (the set of labels in CL violates some constraints) DO
    BEGIN
      <x,v> ← a randomly picked label from CL which is incom-
        patible with some other labels in CL;
      v' ← any value in D_x such that CL – {<x,v>} + {<x,v'>} vio-
        lates no more constraints than CL;
      CL ← CL – {<x,v>} + {<x,v'>};
    END
END /* of Naive_CSP_Hill_Climbing */
```

The Naive_CSP_Hill_Climbing algorithm continues to iterate in the WHILE loop as long as it can pick a random label that is in conflict with some other labels in the current compound label CL. For the label picked, it revises the label by picking a value that violates no more constraints than the original one (this allows the picking of the current value). This algorithm terminates if and when the current compound label is a solution (i.e. it violates no constraints).

The problem with hill-climbing algorithms in general is that they do not guarantee successful termination. They may settle in local optima, where all adjacent points are worse than the current focal point, though the current focal point does not represent a solution (this will not happen in Naive_CSP_Hill_Climbing). They may also loop in *plateaus*, where a number of mutually-adjacent points all have the same value (see Figure 8.1). Sometimes, additional termination conditions are added (to the WHILE loop in the Hill_Climbing algorithm above). For example, one may want to limit the number of iterations or the program's run time.

Even when hill-climbing algorithms terminate, they are not guaranteed to be efficient. But when good heuristics are available, which could be the case in some problems, hill-climbing does give us hope to solve intractable CSPs.

### 8.2.2 The heuristic repair method

The *heuristic repair method* is a hill-climbing method based on the min-conflict heuristic described in Section 6.3.2 of Chapter 6. It improves over the Naive_CSP_hill_climbing algorithm in the way in which it chooses the values. When a label which violates some constraints is picked for revision, the value which violates the least number of constraints is picked. Ties are resolved randomly. The pseudo code for the Heuristic Repair Method is shown below:

```
PROCEDURE Heuristic_Repair(Z, D, C)
/* A hill-climbing algorithm which uses the Min-Conflict heuristic */
BEGIN
```

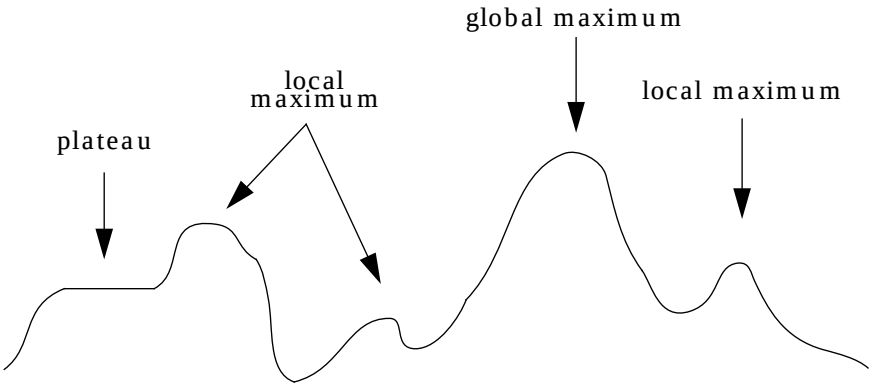



**Figure 8.1** Possible problems with hill-climbing algorithms: the algorithms may stay in plateaus or local maxima

```
/* Part 1: Initialization */
CL ← { };
FOR each x ∈ Z DO
    BEGIN
        V ← the set of values in D_x which violate the minimum
            number of constraints with labels in CL;
        v ← a random value in V;
        CL ← CL + {<x,v>};
    END
/* Part 2: Hill-climbing */
WHILE (the set of labels in CL violates some constraints) DO
/* additional termination conditions may be added here */
    BEGIN
        <x,v> ← a randomly picked label from CL which is incom-
            patible with some other labels in CL;
        CL ← CL – {<x,v>};
        V ← the set of values in D_x which violates the minimum
            number of constraints with the other labels in CL;
        v' ← random value in V;
        CL ← CL + {<x,v'>};
    END
END /* of Heuristic_Repair */
```

The Heuristic Repair Method has been applied to the *N*-queens problem. The one-million-queens problem is reported to have been solved by the Heuristic Repair Method in less than four minutes (real time) on a SUN Sparc 1 workstation. It should be reiterated here that results in testing an algorithm on the *N*-queens problem may be deceptive, because the *N*-queens problem is a very special CSP in which the binary constraints become looser as *N* grows.

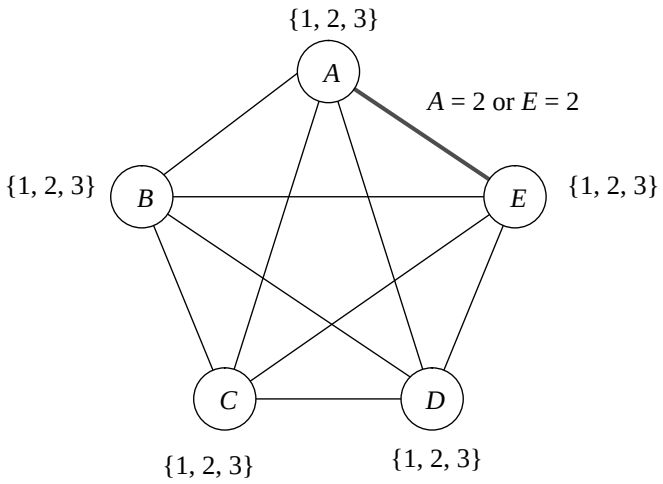
Program 8.1, *hc.plg*, shows an implementation of the Heuristic Repairs Method. This program does not guarantee to find solutions in the *N*-queens problem.

The Heuristic Repair Method has the usual problem of incompleteness in hill-climbing algorithms. The example in Figure 8.2 shows a problem in which the Heuristic Repair Method would fail to produce a solution in most attempts. This problem contains five variables, *A* to *E*, whose domains are all {1, 2, 3}. There exists only one solution, which is to have all variables labelled to 2. The Heuristic Repair Method will normally fail to find this solution because unless three or more variables are initialized to 2, most variables will end up with values 1 or 3 and the Heuristic Repair Method will wander around in a plateau of local minima. For example, assume that the initialized compound label is (<*A*,2><*B*,1><*C*,1><*D*,1><*E*,2>). Six constraints, namely $C_{A,B}$, $C_{A,C}$, $C_{A,D}$, $C_{B,E}$, $C_{C,E}$ and $C_{D,E}$, are violated. The number of constraints violated can be reduced if the value of either *A* or *E* is changed to 1 or 3 (in either case, only four constraints will be violated). Even if one of *B*, *C* or *D* is picked to have its value revised, changing its value to 2 does not reduce the number of constraints violated, and therefore, there is a 2/3 chance that the values 1 or 3 will be picked (which does not bring the algorithm closer to the solution). If the initialized compound label has two or less 2's assigned to the variable, and *A* and *E* are not both labelled with 2, e.g. (<*A*,2><*B*,2><*C*,1><*D*,1><*E*,1>), then the Heuristic Repair Method will change between states in which the five variables take values 1 or 3, which always violate one constraint, namely $C_{A,E}$.

### 8.2.3 A gradient-based conflict minimization hill-climbing heuristic

The *gradient-based conflict minimization* (*GBCM*) *heuristic* is one that is applicable to CSPs where all variables have the same domain and the size of this domain is the same as the number of variables, and where each variable must take a unique value. It has been found to be effective in the *N*-queens problem, although its effectiveness in other problems is unknown. Since the *N*-queens problem has been used to illustrate many algorithms in this book, we shall include this heuristic here for the sake of completeness.

Like the Naive_CSP_Hill_Climbing algorithm, an algorithm which uses the GBCM heuristic hill-climbs from a random compound label. If there exist two labels in the compound label which are in conflict with other labels, the values of them will be swapped when the compound label after this swap violates fewer constraints. The




(All constraints $C_{xy}$, with the exception *of* $C_{AE}$, require that $x + y$ is even; $C_{AE}$ requires that at least one of $A$ and $E$ takes the value 2)

**Figure 8.2** Example of a CSP in which the Heuristic Repair Method would easily fail to find the only solution where all variables are assigned the value 2

idea is similar to the *2-opting heuristic* in the travelling salesman problem (see, for example, Aho *et al*., 1983). The algorithm, called QS1 and designed for solving the *N*-queens problem, is shown below:

```
PROCEDURE QS1(n)
/* n is the number of queens in the N-queens problem */
BEGIN
    /* initialization */
    FOR i = 1 to n DO
        Q[i] ← a random column which is not yet occupied;
    /* hill-climbing */
```

```
    WHILE conflict exists DO
        BEGIN
            find any i, j, such that Q[i], Q[j] are in conflict with some
                queens;
            IF (swapping values of Q[i], Q[j] reduces the total number of
                conflicts)
            THEN swap the values of Q[i] and Q[j];
        END /* of while loop */
END /* of QS1 */
```

It is found that the initialization part of the QS1 algorithm can be improved. Firstly, a constant $c$ is chosen. Then $n - c$ random rows are chosen, and a queen is put into one column of each row, making sure that no queens attack each other. If no safe column is found in any of the chosen rows, then this row is replaced by another random row. There is no backtracking involved. After initialization, the program proceeds in the same way as QS1. The resulting program is called QS4:

```
PROCEDURE QS4(n)
    CONSTANT: c;        /* c is to be determined by the programmer */
BEGIN
    /* initialization — minimize conflicting queens */
    FOR i = 1 to n – c DO
        place a queen in a random position which does not have con-
            flict with any queen which has already been placed; if failed,
            exit reporting failure;
    FOR i = 1 to c DO
        place a queen in a random column which is not yet occupied;
    /* hill-climbing */
    WHILE (conflict exists) DO
        BEGIN
            find any i, j, such that Q[i], Q[j] are in conflict with some
                queens;
            IF (swapping values of Q[i], Q[j] reduces the total number of
                conflicts)
            THEN swap the values of Q[i] and Q[j];
        END /* of while loop */
END /* of QS4 */
```

It is found that with a properly chosen $c$, QS4 performs better than the Heuristic Repair Method. For the one-million-queens problem, QS4 takes an average of 38 CPU seconds on a SUN Sparc 1 workstation, while the Heuristic Repair Method takes 90-240 seconds. Solutions are found for the three-million-queens problem in 54.7 CPU seconds.

However, it is unclear how effective the GBCM heuristic is in problems other than the *N*-queens problem. Apart from the limitation that it is only applicable to problems in which each variable must take a different value, the choice of *c* in QS4 is very important. If *c* is too small, QS4 shows no improvement over QS1. If *c* is too large, the initialization process may fail (since no backtracking is involved). No mechanism has been proposed for choosing *c*.

## 8.3 Connectionist Approach

### 8.3.1 Overview of problem solving using connectionist approaches

The *min-conflict heuristic* described above (Sections 6.3.2 and 8.2.2) is derived from a connectionist approach. A connectionist approach uses networks where the nodes are very simple processors and the arcs are physical connections, each of which is associated with a numerical value, called a *weight*. At any time, each node is in a state which is normally limited to either *positive* (*on*) or *negative* (*off*). The state of a node is determined locally by some simple operations, which take into account the states of this node's directly connected nodes and the weights of those connecting arcs. The *network state* is the collection of the states of all the individual nodes. In applying a connectionist approach to problem solving, the problem is represented by a connectionist network. The task is to find a network state which represents a solution.

Connectionist approaches to CSPs have attracted great attention because of their potential for massive parallelism, which gives hope to the solving of problems that are intractable under conventional methods, or of solving problems with a fraction of the time required by conventional methods, sometimes at the price of losing completeness. A connectionist approach for maintaining arc-consistency has been described in Section 4.7 of Chapter 4. In this section, one connectionist approach to CSP solving is described.

### 8.3.2 GENET, a connectionist approach to the CSP

GENET is a connectionist model for CSP solving. It has demonstrated its effectiveness in binary constraint problems, and is being extended to tackle general CSPs. In this section, we shall limit our attention to its application to binary CSPs. Given a binary CSP, each possible label for each variable is represented by a node in the connectionist network. All the nodes for each variable are collected to form a cluster. Every pair of labels between different clusters which is prohibited by a constraint is connected by an inhibitory link. Figure 8.3 shows an example of a CSP (a simplified version of the problem in Figure 8.1) and its representation in GENET. For example, $A + B$ must be even, and therefore (<*A*,1><*B*,2>) is illegal; hence the

nodes which represent <*A*,1> and <*B*,2> are connected. $C_{A,E}$ requires $A = 2$, $E = 2$ or both to be true. Consequently, there are connections between <*A*,1> and both <*E*,1> and <*E*,3>, and connections between <*A*,3> and both <*E*,1> and <*E*,3>.

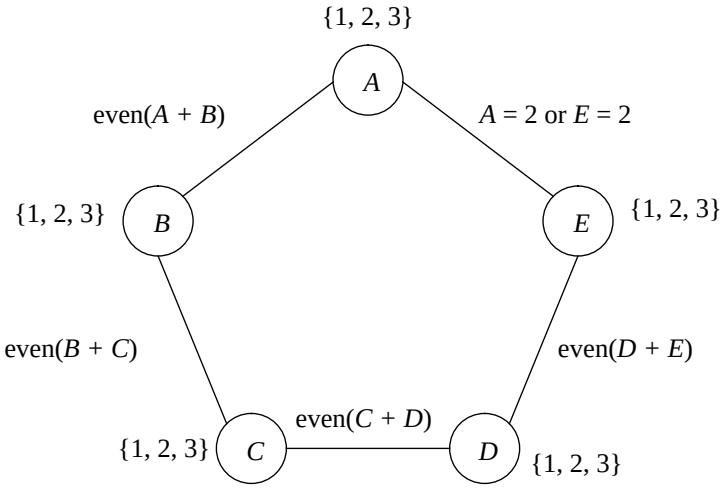
The algorithm of GENET is very simple. The network is initialized by assigning -1 to all the weights. One arbitrary node per cluster is switched on, then the network is allowed to converge under the rule which we shall describe below. The input to each node is computed by the following rule:

$$\text{input of } x = \sum_{y \leftarrow adjacent\,(x,\,y)} w_{x,\,y} \times s_y$$

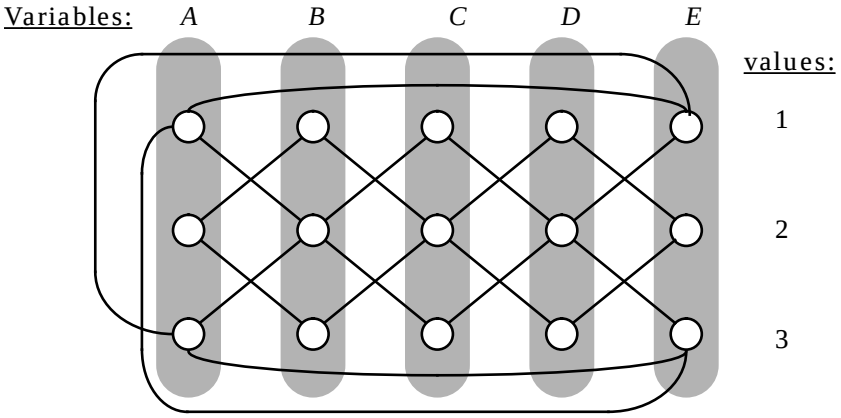
where $w_{x,y}$ is the weight of the connection between nodes $x$ and $y$, and $s_y$ is the state of $y$, which is 1 if $y$ is on and 0 if $y$ is off. That means that the input to a node $x$ is the sum of the weights on the connections which connect $x$ to nodes that are *on* at the point of calculation. The nodes in each cluster continuously compete to be turned on. In every cluster, the node that receives the maximum input will be turned on, and the rest will be turned off. Since there exist only connections with negative weights, the winner in each cluster represents a label which violates the fewest constraints for the subject variable. In tie situations, if one of the nodes in the tie was already on in the previous cycle, it will be allowed to stay on. If all the nodes in the tie were off in the previous cycle, then a random choice is made to break the tie. (Experiments show that breaking ties randomly, as done in the heuristic repair method, degrades the performance of GENET.)

Figure 8.4 shows a state of the network shown in Figure 8.3. There, the nodes which are *on* are highlighted, and the input is indicated next to each node. The cluster of nodes which represent the labels for variable *A* is unstable because the node which represents <*A*,2> has the highest input (0) but is not switched on. Similarly, the clusters for *B* and *C* are unstable because the *on* nodes in them do not have the highest input. Cluster *D* is stable because the node which represents <*D*,2> has an input of −1, which is a tie with the other two nodes in cluster *D*. According to the rules described above, the node representing <*D*,*E*> will remain *on*. There is no rule governing which of the clusters *A*, *B* or *C* should change its state next, and this choice is non-deterministic. Obviously, the change of state in one cluster would change the input to the nodes in other clusters. For example, if the node for <*A*,1> is switched *off*, and the node for <*A*,2> is switched *on*, the input of all the three nodes in cluster *B* would be −1, which means cluster *B* would become stable.

If and when the network settles in a stable state, which is called a *converged* state, GENET will check to see if that state represents a solution. In a converged state, none of the *on* nodes have lower input than any other nodes in the same cluster. Figure 8.5 shows a converged state in the network in Figure 8.3. A state in which all the *on* nodes have zero input represents a solution. Otherwise, the network state rep-




(a) Example of a binary CSP (variables: *A*, *B*, *C*, *D* and *E*)




(b) Representation of the CSP in (a) in GENET all connections have their weights initialized to –1

**Figure 8.3** Example of a binary CSP and its representation in GENET

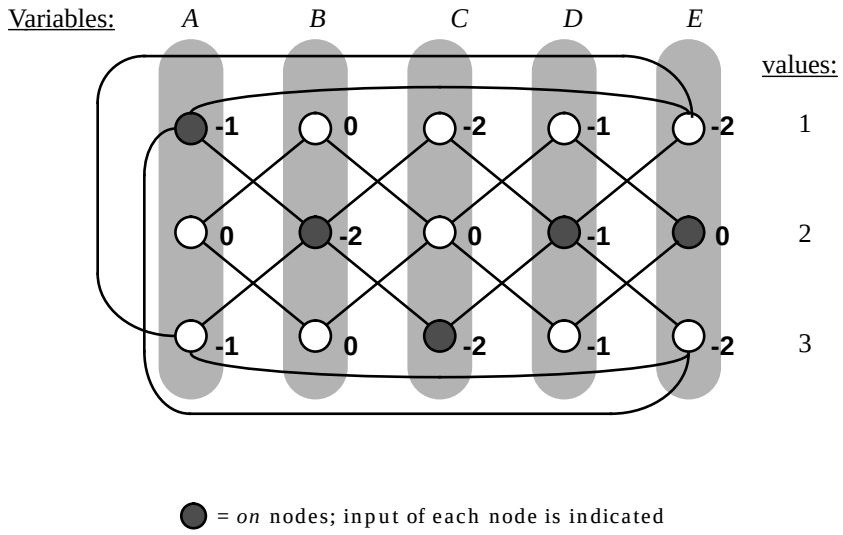



**Figure 8.4** Example of a network state in the GENET network shown in Figure 8.3(b) (all connections have weights equal to −1)

resents a local minimum. The converged state in Figure 8.5 represents a local minimum because the inputs to the nodes which represent <*D*,1> and <*E*,2> are both −1.

When the network settles in a local minimum, the state updating rule has failed to use local information to change the state. When this happens, the following heuristic rule is applied to remove local maxima:

$$\text{New } w_{ij} = \text{Old } w_{ij} + s_i \times s_j$$

The local maxima is removed by decreasing the weights of violated connections (constraints). This simple "learning" rule effectively does two things: it reduces (continuously if necessary) the value of the current state until it ceases to be a local minima. Besides, it reduces the possibility of any violated constraint being violated again. The hope is that after sufficient "learning" cycles, the connection weights in the network will lead the network states to a solution.

In the example in Figure 8.5, the weight on the connection between the nodes which

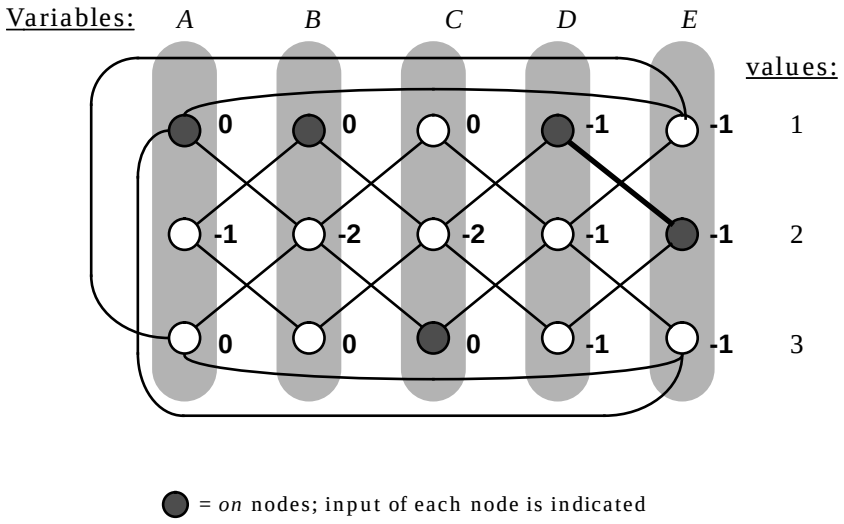



**Figure 8.5** Example of a converged state in the GENET network shown in Figure 8.3(b) (all connections have weights equal to −1)

represent <*D*,1> and <*E*,2> (highlighted in Figure 8.5) will be decreased by 1 to become −2. This will make the input to both of the nodes for <*D*,1> and <*E*,2> −2. As a consequence, the state of either cluster *D* or cluster *E* will be changed.

The GENET algorithm is shown below in pseudo code:

```
PROCEDURE GENET
BEGIN
    One arbitrary node per cluster is switched ON;
    REPEAT
        /* network convergence: */
        REPEAT
            Modified ← False;
            FOR each cluster C DO IN PARALLEL
                BEGIN
                    On_node ← node in C which is at present ON;
```

```
                Label_Set ← the set of nodes within C which input
                    are maximum;
                IF NOT (On_node in Label_Set) THEN
                    BEGIN
                        On_node ← OFF;
                        Modified ← True;
                        Switch an arbitrary node in Label_Set to ON;
                    END
            END
        UNTIL (NOT Modified);       /* the network has converged */
        /* learn if necessary: */
        IF (sum of input to all ON nodes < 0)
                /* network settled in local maximum */
        THEN FOR each connection c connecting nodes x & y DO IN
            PARALLEL
                    IF (both x and y are ON)
                    THEN decrease the weight of c by 1;
    UNTIL (input to all ON nodes are 0) OR (any resource exhausted)
END /* of GENET */
```

The states of all the nodes are revised (and possibly updated) in parallel asynchronously in this model. The inner REPEAT loop terminates when the network has converged. The outer REPEAT loop terminates when a solution has been found, or some resource is exhausted. This could mean that the maximum number of cycles has been reached, or that the time limit of GENET has been exceeded.

### 8.3.3 Completeness of GENET

There is no guarantee of completeness in GENET, as can be illustrated by the simple example in Figure 8.6.

The problem in Figure 8.4 comprises two variables, $A$ and $B$, in which the domains are both $\{1, 2\}$. A constraint between $A$ and $B$ requires them to take values of which the sum is odd. Therefore, nodes which represent $\langle A,1\rangle$ and $\langle B,1\rangle$ are connected, and nodes which represent $\langle A,2\rangle$ and $\langle B,2\rangle$ are connected in GENET.

GENET may not terminate in this example because the following scenario may take place: the network is initialized to represent ($\langle A,1\rangle$ $\langle B,1\rangle$). Then, since inputs to both of the nodes which represent $\langle A,1\rangle$ and $\langle B,1\rangle$ are $-1$, and as inputs to both of the nodes which represent $\langle A,2\rangle$ and $\langle B,2\rangle$ are 0, both clusters will change state. If both clusters happen to change states simultaneously at all times, then the network will oscillate between the states which represent ($\langle A,1\rangle$ $\langle B,1\rangle$) and ($\langle A,2\rangle$ $\langle B,2\rangle$) and never converge.

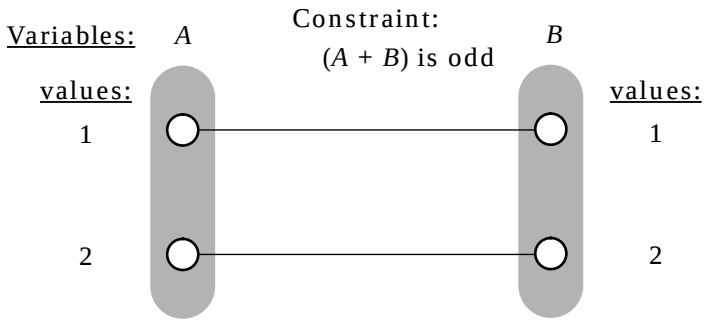



**Figure 8.6** Example of a network in GENET which may not converge (the network may oscillate between (<*A*,1> <*B*,1>) and (<*A*,2> <*B*,2>))

### 8.3.4 Performance of GENET

A simulator of GENET has been implemented. Within the simulator, the clusters are revised sequentially in the procedure shown above. The simulator is allowed a limited number of state changes, and when the limit is exceeded the simulator will report failure. The result of GENET is compared with a program which performs complete search by using forward checking (Section 5.3.1) and the fail-first principle (Section 6.2.3) to check if the simulator has missed any solution and to evaluate the speed of GENET.

Thousands of tests have been performed on the GENET simulator using designed and randomly generated problems. Local minima are known to be present in the designed problems (the problem in Figure 8.2 being one example). Binary CSPs are randomly generated using the following parameters:

$N$ = number of variables;
$d$ = average domain size;
$p_1$ = the probability of two variables being constrained to each other;
$p_2$ = the probability of two labels being compatible with each other in a given constraint.

Parameters have been chosen carefully in generating the random problems so as to

focus on tight problems (where relatively few solutions exist), as they are usually those problems which are most difficult to solve by stochastic methods. Although GENET does not guarantee completeness, the simulator has not missed any solution within 1000 cycles in all the CSPs tested so far. This gives positive support to the hypothesis that GENET will only miss solutions in a relatively small proportion of problems.

The potential of GENET should be evaluated by the number of cycles that it takes to find solutions. For CSPs with $N = 170$, $d = 6$, $p_1 = 10\%$ and $p_2 = 85\%$, GENET takes just over 100 cycles to find solutions when they exist. As a rough estimation, an analogue computer would take $10^{-8}$ to $10^{-6}$ seconds to process one cycle. Therefore, if GENET is implemented using an analogue architecture, then we are talking about spending something like $10^{-6}$ to $10^{-4}$ seconds to solve a CSP with $6^{170}$ states to be searched. To allow readers to evaluate this speed, the complete search program mentioned above takes an average of 45 CPU minutes to solve problems of this size (average over 100 runs). This program implements forward checking and the fail first principle in C, and timing obtained by running it on SUN Sparc1 workstations. The efficiency of this program has to be improved $10^7$ times if it is to match the expected performance of the target GENET connectionist hardware.

## 8.4 Summary

In some applications, the time available to the problem solver is not sufficient to tackle the problem (which involves either finding solutions for it or concluding that no solution exists) by using complete search methods. In other applications, delay in decisions could be costly. Stochastic search methods, which although they do not normally guarantee completeness, may provide an answer to such applications. In this chapter, two stochastic search techniques, namely hill-climbing and connectionist approaches, have been discussed. Preliminary analysis of these techniques gives hope to meeting the requirements of the above applications.

Search strategies that we have described so far normally start with an empty set of assignments, and add one label to it at a time, until the set contains a compound label for all the variables which satisfy all the constraints. So their search space is made up of *k*-compound labels, with *k* ranging from 0 to *n*, where *n* is the number of variables in the problem. On the contrary, the hill-climbing and connectionist approaches search the space of *n*-compound labels.

The heuristic repair method is a hill-climbing approach for solving CSPs. It uses the min-conflict heuristic introduced in Chapter 6. Starting with an *n*-compound label, the heuristic repair method tries to change the labels in it to reduce the total number of constraints violated. The gradient-based conflict minimization heuristic is another heuristic applicable to CSPs where all the *n* variables share the same

domain of size *n*, and each variable must take a unique value from this domain. Starting from an *n*-compound label, the strategy is to swap the values between pairs of labels so as to reduce the number of constraints being violated. Both of these heuristics have been shown to be successful for the *N*-queens problem. Like many other hill-climbing strategies, solutions could be missed by algorithms which adopt these heuristics.

GENET is a connectionist approach for solving CSPs. A given CSP is represented by a network, where each label is represented by a node and the constraints are represented by connections among them. Each state of the network represents an *n*-compound label. Associated with each connection is a weight which always take negative values. The nodes in the network are turned on and off using local information — which includes the states of the nodes connected to it and the weights of the connections. The operations are kept simple to enable massive parallelism. Though completeness is not guaranteed, preliminary analysis shows that solutions are rarely missed by GENET for binary CSPs. Hardware implementation of GENET may allow us to solve CSPs in a fraction of the time required by complete search algorithms discussed in previous chapters.

## 8.5 Bibliographical Remarks

Research on applying stochastic search strategies to problem solving is abundant. In this chapter, we have only introduced a few which have been applied to CSP solving. The *heuristic repair method* is reported in Minton *et al*. [1990, 1992]. It is a domain independent algorithm which is derived from Adorf & Johnston's [1990] neural-network approach. Sosic & Gu [1991] propose QS1 and QS4 for solving the *N*-queens problem more efficiently, by exploiting certain properties of the problem. QS4 is shown to be superior to the heuristic repair method, but the comparison between QS1 and the heuristic repair method has not been reported. (As mentioned in Chapter 1, the *N*-queens problem is a very special CSP. By exploiting more properties of the *N*-queens problem, Abramson & Yung [1989] and Bernhardsson [1991] solve the *N*-queens problem without needing any search.) Smith [1992] uses a min-conflict-like reassignment algorithm for loosely constrained problems. Another generic greedy hill-climbing strategy is proposed by Selman *et al*. [1992]. Morris [1992] studies the effectiveness of hill-climbing strategies in CSP solving, and provides an explanation for the success of the heuristic repair method.

Saletore & Kale [1990] support the view that linear speed up is possible using multiple processors. However, Kasif [1990] points out that even with a polynomial number of processors, one is still not able to contain the combinatorial explosion problem in CSP. Collin *et al*. [1991] show that even for relatively simple constraint graphs, there is no general model for parallel processing which guarantees completeness.

GENET is ongoing research which uses a connectionist approach to CSP solving. The basic model and preliminary test results of GENET are reported by Wang & Tsang [1991]. (The random CSPs are generated using the same parameters as those in Dechter & Pearl [1988a].) Tsang & Wang [1992] outline a hardware design to show the technical feasibility of GENET.

Connectionist approaches to arc-consistency maintenance, including work by Swain & Cooper [1988, 1992] and Guesgen & Hertzberg [1991, 1992], have been discussed in Chapter 4. Guesgen's algorithm is not only sound and complete, but is also guaranteed to terminate. However, when the network converges, what we get is no more than a reduced problem which is arc-consistent, plus some additional information which one could use to find solutions. The task of generating solutions from the converged network is far from trivial.

Literature on connectionism is abundant; for example, see Feldman & Ballard [1982], Hopfield [1982], Kohonen [1984], Rumelhart *et al*. [1986], and Grossberg [1987]. Partly motivated by the CSP, Pinkas & Dechter [1992] look at acyclic networks.

Closely related to hill-climbing and connectionist approaches is *simulated annealing*, whose full potential in CSP solving is yet to be explored. For reference to simulated annealing see, for example, Aarts & Korst [1989], Davis [1987] and Otten & van Ginneken [1989].